Task Partitioning

introduction

- Task partitioning strategy in parallel computing system is the key factor to decide the efficiency, speedup of the parallel computing systems. The process is partitioned into the subtasks where the size of the task is determined by the run time performance of the each server

System Partitioning

- The functionality of a system is implemented with a set of interconnected system components, such as ASIC's, memories, CPU's, buses.

- The designer must solve two problems:
  - select a set of system components (**allocation**),
  - partition the system's functionality among these components (**partitioning**).

- The final implementation has to satisfy a set of design constraints, such as:
  - **cost,**
  - **performance and**
  - **power consumption**

Structural Partitioning

- First the system components are implemented using interconnected hardware components.
- Partitioning separates the objects into groups, where each group represents a system component.
- Mostly used at lower levels of abstraction for hardware partitioning.
- Satisfies certain constraints (for instance packaging).
- Problems:
  - **size/performance tradeoffs are difficult,**
  - **large number of objects**.

Functional Partitioning

- The system level *functionality* is partitioned in order to divide the behavior of the system between multiple components.
- Usually executable model is partitioned and therefore the estimation of parameters and partitioning results is possible.
- Advantages:
  - **size/performance tradeoffs,**
  - **small number of objects,**
  - **hardware/software solutions.**

Types of Parallelism : Two Extremes

- Data parallel
  - Each processor performs the same task on different data
  - Example - grid problems
- Task parallel
  - Each processor performs a different task
  - Example - signal processing
- Most applications fall somewhere on the continuum between these two extremes

Basics of Data Parallel Programming

- One code will run on 2 CPUs
- Program has array of data to be operated on by 2 CPU so array is split into two parts.

```
program:
…
if CPU=a then
   low_limit=1
   upper_limit=50
elseif CPU=b then
   low_limit=51
   upper_limit=100
end if
do I = low_limit,
upper_limit
   work on A(I)
end do
...
end program
```

CPU A

```
program:
…
low_limit=1
upper_limit=50
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

CPU B

```
program:
…
low_limit=51
upper_limit=100
do I= low_limit,
upper_limit
   work on A(I)
end do
…
end program
```

Basics of Task Parallel Programming

- One code will run on 2 CPUs
- Program has 2 tasks (a and b) to be done by 2 CPUs

```
program.f:
…
initialize
...
if CPU=a then
   do task a
elseif CPU=b then
   do task b
end if
….
end program
```

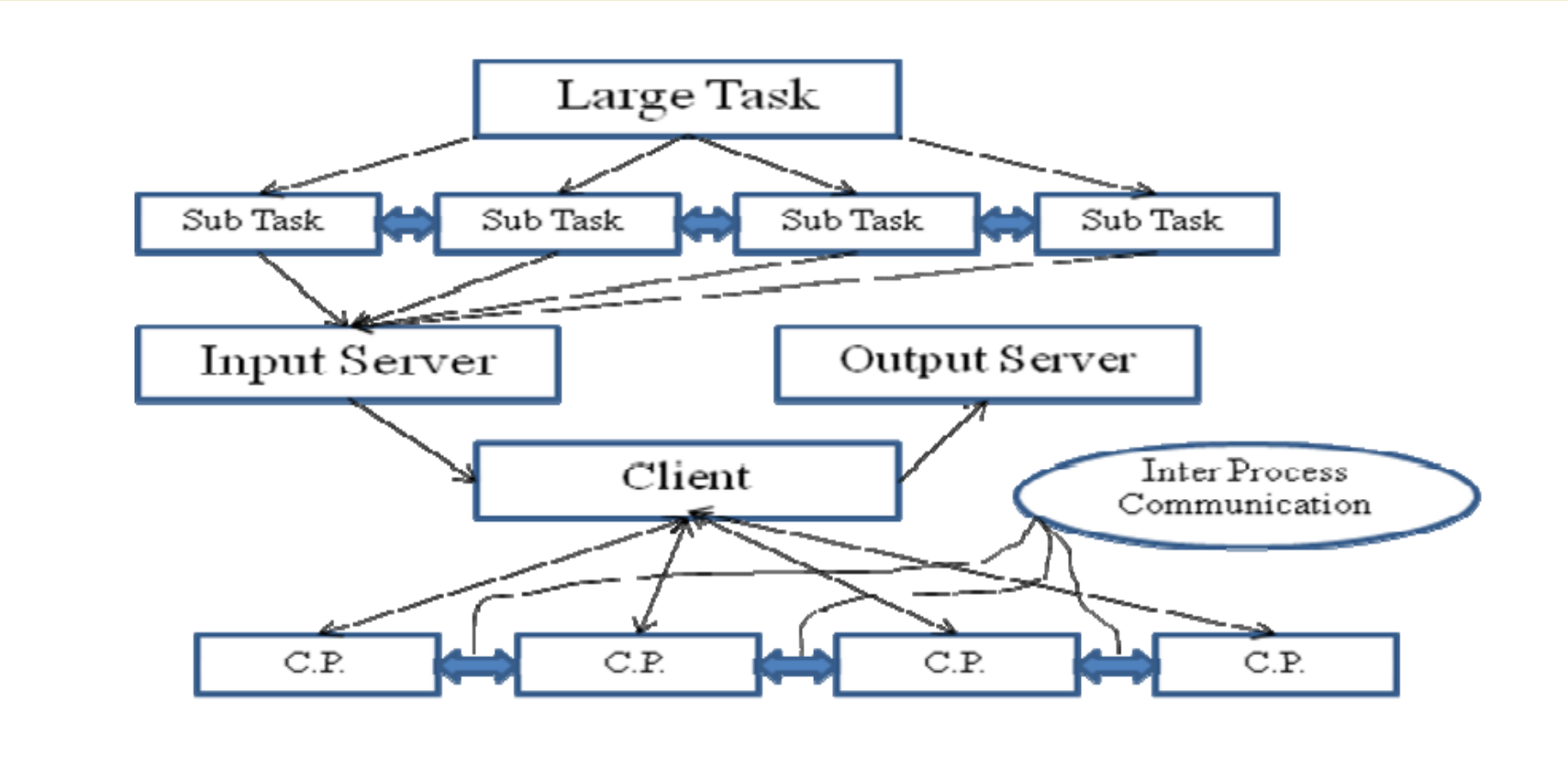
CPU A

```
program.f:
…
initialize
…
do task a
…
end program
```

CPB

```
program.f:
…
initialize
…
do task b
…
end program
```

Task partitioning strategy

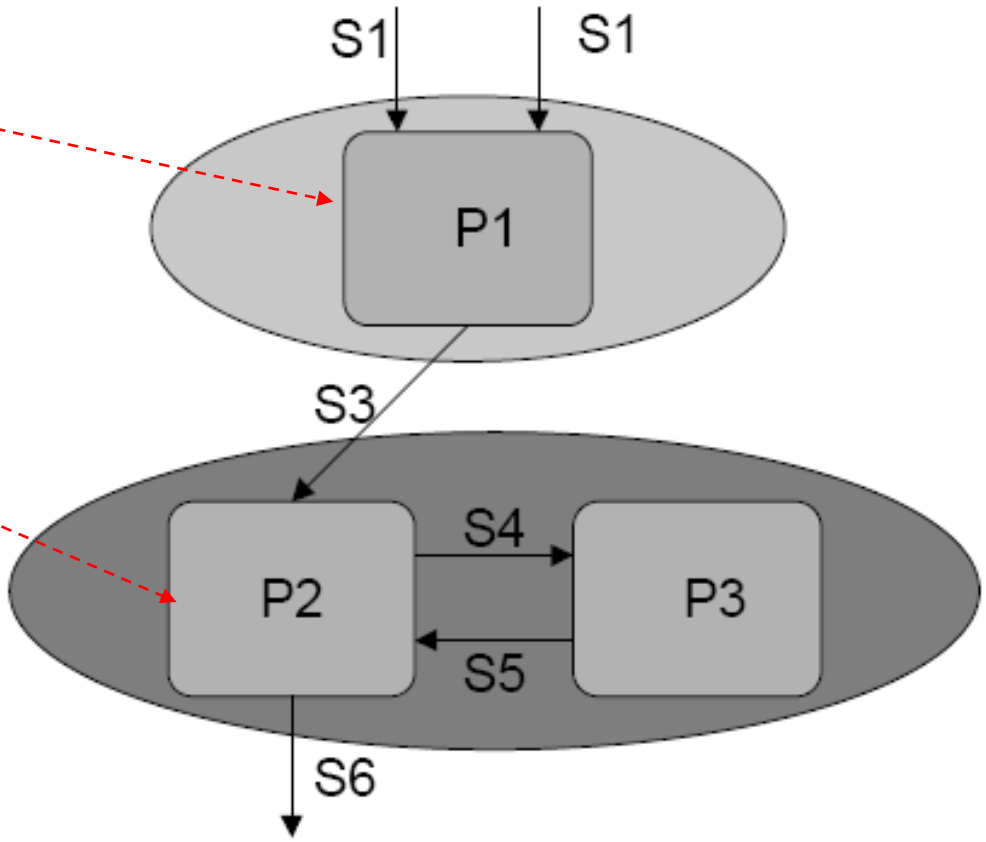


Task Partitioning

```
. . .
signal S1, S2, S3, S4, S5, S6: INTEGER;
. . .
P1: process
        variable A, B: INTEGER;
begin
        . . .
        A:=(S1+5)*3;
        B:=S1+S2+7;
        S3<=A*B;
        . . .
end process;

P2: process
        variable X, Y: INTEGER;
begin
        . . .
        wait on S3;
        S4<=S3+X;
        . . .
        wait on S5;
        S6<=S5*Y;
end process;

P3: process
        variable Z: INTEGER;
begin
        wait on S4;
        . . .
        S5<=S4+Z;
```



Metrics and Estimations

- Partitioning algorithms have to rely on a quantitative measure of a candidate solution's goodness.

- ***Metrics*** — attributes which characterize a given solution;
  - they are expressed quantitatively.

- ***Metrics*** include:
  - cost,
  - execution time,
  - communication rates,
  - power consumption,
  - testability,
  - reliability,
  - program size,
  - data size
  - and memory size.

Metrics and Estimations

- Estimation determines a metric value from a *rough implementation*.
- Inaccuracy can be tolerated as long as the *relative goodness* of any two partitions is determined correctly.

Limits of Parallel Computing

- Theoretical Upper Limits
  - Amdahl's Law
- Practical Limits
  - Load balancing
  - Non-computational sections
- Other Considerations
  - time to re-write code

Theoretical Upper Limits to Performance

- All parallel programs contain:
  - Serial sections
  - Parallel sections
- Serial sections limit the parallel effectiveness
- Speedup is the ratio of the time required to run a code on one processor to the time required to run the same code on multiple (N) processors
- Amdahl's Law states this formally

Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.
  - Effect of multiple processors on run time

$$t_n = \left(f_p / N + f_s\right)\ t_1$$

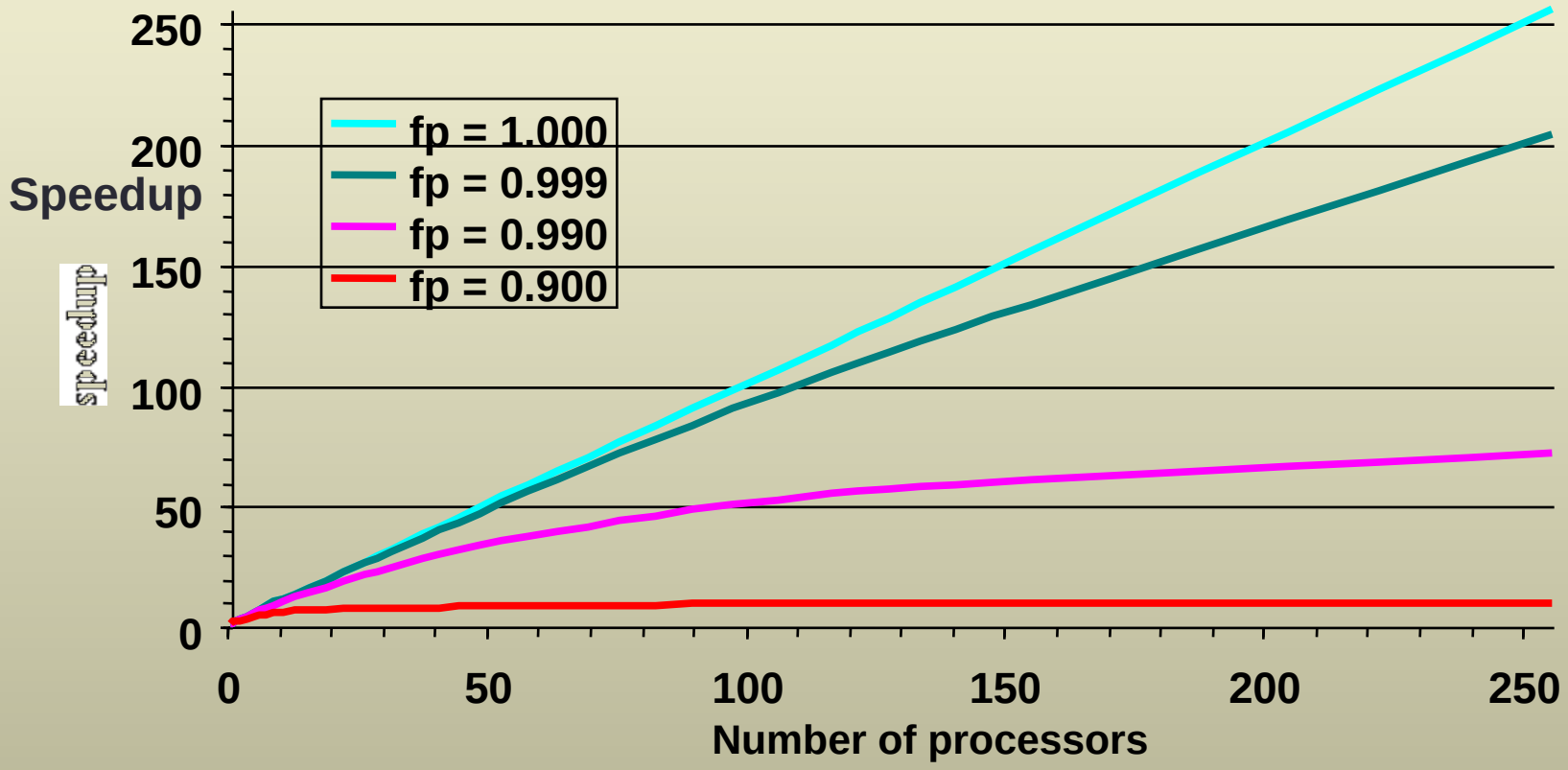
  - Effect of multiple processors on speed up
  - Where

$$S = \frac{1}{f_s + f_p / N}$$

    - fs = serial fraction of code
    - fp = parallel fraction of code
    - N = number of processors
    - t_n = time to run on N processors

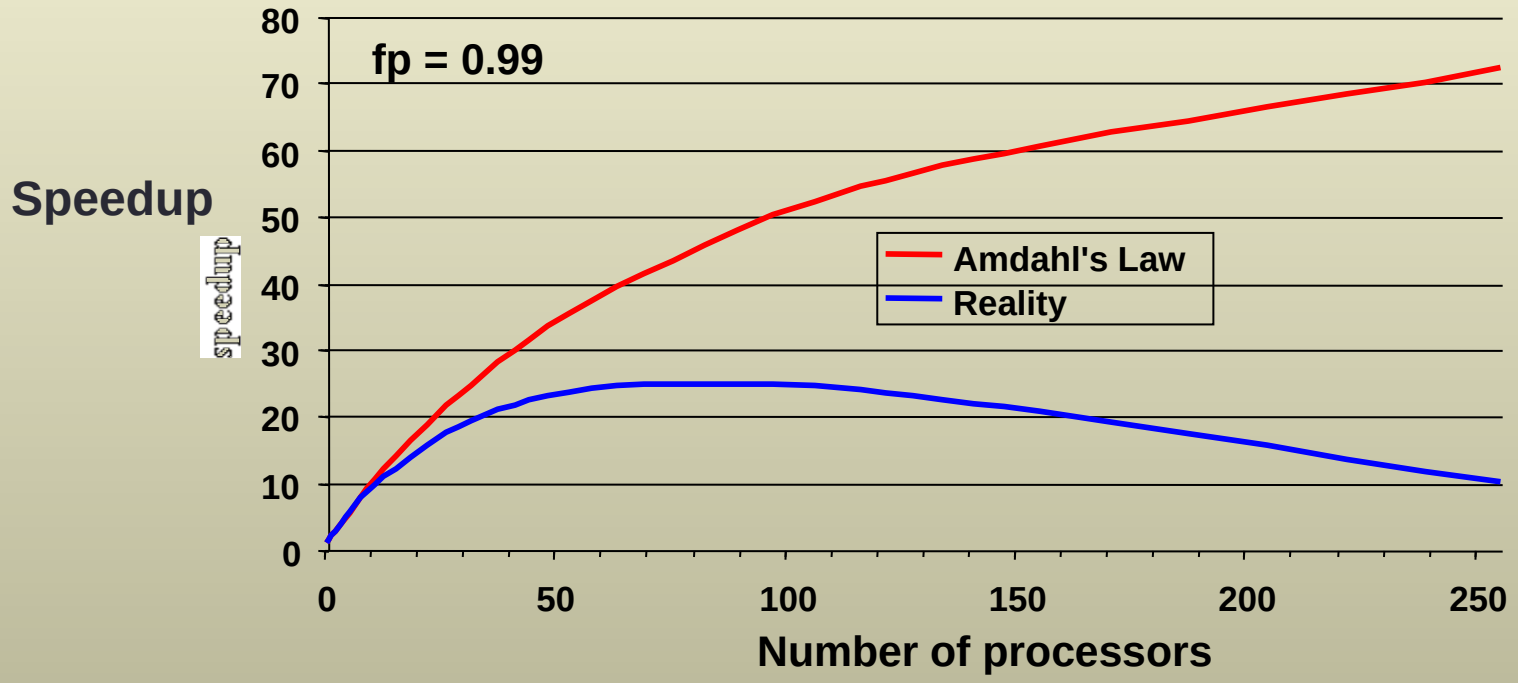
Illustration of Amdahl's Law

It takes only a small fraction of serial content in a code to degrade the parallel performance.



Practical Limits: Amdahl's Law vs. Reality

Amdahl's Law provides a theoretical upper limit on parallel speedup assuming that there are no costs for speedup assuming that there are no costs for *communications*. In reality, communications will result in a further degradation of performance.



QUESTIONS ?

THANK YOU ALL…… !!!